**Linear Search and Binary Search**

The input is an array $A$ of elements in any arbitrary order and a key $k$ and the objective is to output true, if $k$ is in $A$, false, otherwise. Below is a recursive function to solve this problem.

```python
def LinearSearch(A[lo .. hi], k):
    if lo > hi:
        return False
    else:
        return (A[hi] == k) or LinearSearch(A[lo..hi-1], k)
```

The recurrence relation to express the running time of `LinearSearch` is given by $T(n) = T(n - 1) + c$, with the base case being $T(0) = 1$. We have already solved this recurrence and it yields a running time of $T(n) = \Theta(n)$.

If the input array $A$ is already sorted, we can do significantly better using binary search as follows.

```python
def BinarySearch(A[lo .. hi], k):
    if lo > hi:
        return False
    else:
        mid = floor(lo+hi/2)
        if A[mid] == k:
            return True
        elif A[mid] < k:
            return BinarySearch(A[mid+1 .. hi], k)
        else:
            return BinarySearch(A[lo .. mid-1], k)
```

The running time of this method is given the recurrence $T(n) = T(n/2) + c$, with the base case being $T(0) = 1$. As we have seen before, this recurrence yields a running time of $T(n) = \Theta(\log n)$.

**Sorting**

Below is a recursive version of insertion sort that we studied a couple of lectures ago.

```python
def InsertionSort(A[lo .. hi]):
    if lo == hi:
        return A
```
else
  A’ = InsertionSort(A[lo..hi-1])
  Insert(A’, A[hi])  // insert element A[hi] into the sorted array A’

Note that the Insert function takes $\Theta(n)$ time for an input array of size $n$. Thus the running time of Insertion sort is given by the following recurrence.

$$T(n) = \begin{cases} 
1, & n = 1 \\
T(n-1) + n, & n \geq 2 
\end{cases}$$

It is easy to see that this recurrence yields a running time of $T(n) = \Theta(n^2)$.

To motivate the idea behind the next sorting algorithm (Merge Sort), let’s rewrite InsertionSort function as follows.

InsertionSort(A[lo..hi])
  if lo = hi then
    return A
  else
    // Merge combines two sorted arrays into one sorted array
    Merge(InsertionSort(A[lo..hi-1]), InsertionSort(A[hi..hi]))

The function Merge is as follows.

Merge(A[1..p], B[1..q])
  if p = 0 then
    return B
  if q = 0 then
    return A
    return prepend(A[1], Merge(A[2..p], B[1..q]))
  else
    return prepend(B[1], Merge(A[1..p], B[2..q]))

Note that the running time of Merge is $O(p+q)$. The second recursive call to InsertionSort takes $O(1)$ time and hence the running time of InsertionSort still is $\Theta(n^2)$.

Observe that in InsertionSort the input array $A$ is partitioned into two arrays, one of size $|A| - 1$ and another of size 1. In Merge Sort, we partition the input array of size $n$ in two equal halves (assuming $n$ is a power of 2). Below is the function.

MergeSort(A[1..n])
  if n = 1 then
    return A
  else
    return Merge(MergeSort(A[1..n/2], MergeSort(A[n/2+1..n])))

The running time of MergeSort is given by the following recurrence.

$$T(n) = \begin{cases} 
1, & n = 1 \\
2T(n/2) + cn, & n \geq 2 
\end{cases}$$

Below are some facts on logarithms that you may find useful.
January 28, 2016  Running time, Divide and Conquer

\[
\begin{align*}
\text{i. } \log_a b &= \frac{1}{\log_b a} \\
\text{ii. } \log_a b &= \frac{\log b}{\log a} \\
\text{iii. } a^{\log_a b} &= b \\
\text{iv. } x^{\log_a x} &= a^{\log_a b}
\end{align*}
\]

We can also solve recurrences by guessing the overall form of the solution and then figure out the constants as we proceed with the proof. Below are some examples.

**Example.** Consider the following recurrence for the **MergeSort** algorithm.

\[
T(n) = \begin{cases} 
1, & n = 1 \\
2T(n/2) + n, & n \geq 2
\end{cases}
\]

Prove that \( T(n) = O(n \lg n) \).

**Solution.** We will first prove the claim by expanding the recurrence as follows.

\[
T(n) = 2T(n/2) + n \\
= 2^2T(n/2^2) + 2n \\
= 2^3T(n/2^3) + 3n \\
\cdots \\
= 2^kT(n/2^k) + kn
\]

The recursion bottoms out when \( n/2^k = 1 \), i.e., \( k = \lg n \). Thus, we get

\[
T(n) = 2^{\lg n}T(1) + n \lg n \\
= \Theta(n \log n)
\]

We will now prove that \( T(n) = O(n \lg n) \) by using strong induction on \( n \). We will show that for some constant \( c \), whose value we will determine later, \( T(n) \leq cn \lg n \), for all \( n \geq 2 \).

**Induction Hypothesis:** Assume that the claim is true when \( n = j \), for all \( j \) such that \( 2 \leq j \leq k \). In other words, \( T(j) \leq cj \lg j \).

**Base Case:** \( n = 2 \). The left hand side is given by \( T(2) = 2T(1) + 2 = 4 \) and the right hand side is \( 2c \). Thus the claim is true for the base case when \( c \geq 2 \).
Induction Step: We want to show that for $k \geq 2$, $T(k + 1) \leq c(k + 1) \log(k + 1)$. We have

$$T(k + 1) = 2T\left(\frac{k + 1}{2}\right) + (k + 1)$$

$$\leq 2c \left(\frac{k + 1}{2} \log\left(\frac{k + 1}{2}\right)\right) + (k + 1)$$

$$= c(k + 1)(\log(k + 1) - \log 2) + (k + 1)$$

$$= c(k + 1)\log(k + 1) - (c - 1)(k + 1)$$

$$\leq c(k + 1)\log(k + 1) \quad \text{(since } c \geq 2)$$

Example. Consider the following recurrence.

$$T(n) = \begin{cases} 
1, & n = 1 \\
2T(n/2) + n^2, & n \geq 2 
\end{cases}$$

Prove that $T(n) = \Theta(n^2)$.

Solution. Clearly, $T(n) = \Omega(n^2)$ (because of the $n^2$ term in the recurrence). To prove that $T(n) = O(n^2)$, we will show using strong induction that for some constant $c$, whose value we will determine later, $T(n) \leq cn^2$, for all $n \geq 1$.

Induction Hypothesis: Assume that the claim is true when $n = j$, for all $j$ such that $1 \leq j \leq k$. In other words, $T(j) \leq cj^2$.

Base Case: $n = 1$. The claim is clearly true as the left hand side and the right hand side, both equal 1.

Induction Step: We want to show that $T(k + 1) \leq c(k + 1)^2$. We have

$$T(k + 1) = 2T\left(\frac{k + 1}{2}\right) + (k + 1)^2$$

$$\leq 2c \left(\frac{k + 1}{2}\right)^2 + (k + 1)^2$$

$$= \left(\frac{c}{2} + 1\right)(k + 1)^2$$

We want the right hand side to be at most $c(k + 1)^2$. This means that we want $c/2 + 1 \leq c$, which holds when $c \geq 2$. Thus we have shown that $T(n) \leq 2n^2$, for all $n \geq 1$, and hence $T(n) = O(n^2)$. 